SSRV RV32IMC CPU CORE

# Introduction

---

## Overview

SSRV CPU core is super-scalar and out-of-order. This 2/3-stage RV32IMC CPU core has outstanding integer processing ability:

- 2-stage: (Dhrystone 2.1 GCC 8.3.0)
    - 2.35 DMIPS/MHz (-O3 –fno-inline compile options)
    - 4.25 DMIPS/MHz (best performance)
- 3-stage: (Dhrystone 2.1 GCC 8.3.0)
    - 2.13 DMIPS/MHz (-O3 –fno-inline compile options)
    - 4.01 DMIPS/MHz (best performance)

The different between 2/3 stages is lack of branch prediction and the latter has to cost 1 extra cycle to flush pipeline.

As for CoreMark benchmark test, there is a comparison between SSRV and SCR1, which is open source from Syntacore. It takes SCR1 3621 ticks for 1 iteration of CoreMark test, but SSRV 2482 ticks(2-stage), 2793 ticks(3-stage). SSRV is 1.3/1.45 times that of SCR1. According to official score of Syntacore, SCR1 is almost 3 CoreMark/MHz. SSRV is deduced to be 4 CoreMark/MHz approximately. Precise CoreMark test will be carried out in the future.

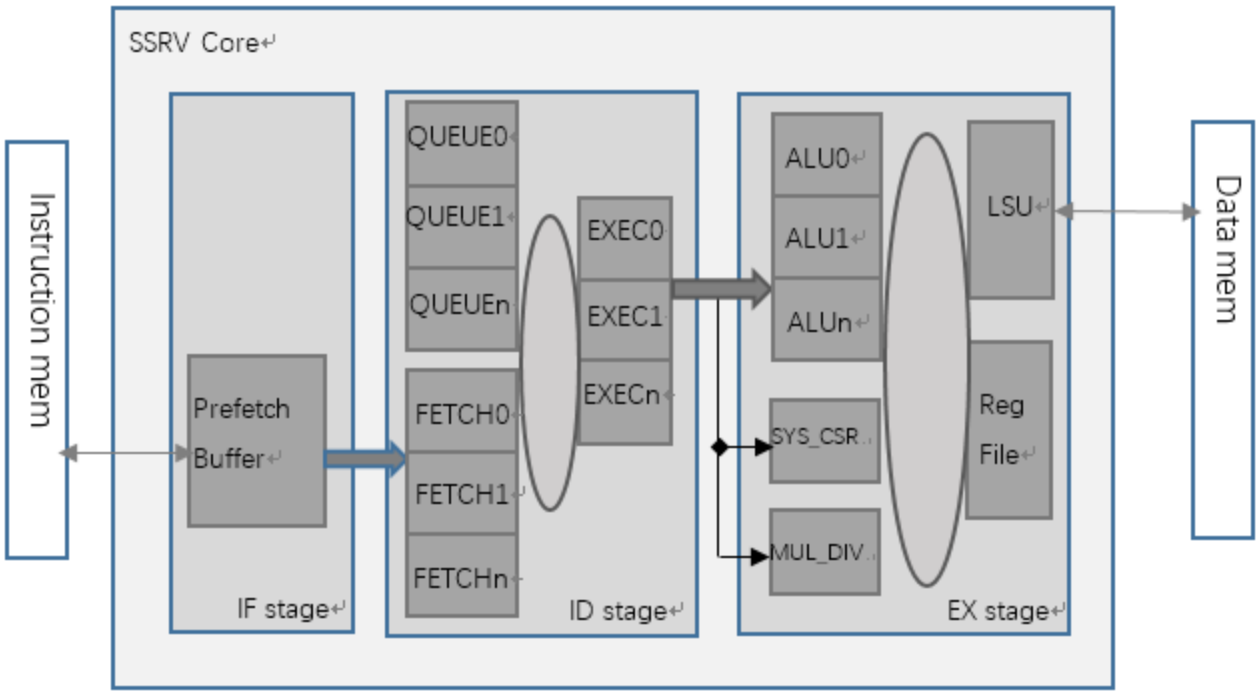
This core can support any number of instructions in parallel. It will estimate instructions fetched to determine how many instructions to be issued actually. Below is parallel-issue status of 16 (1 iteration of " CoreMark 1.0" test):

```
CoreMark 1.0
ticks =      268186  instructions =      282643  I/T = 1.053907
          NUM       TICKS     RATIO
```

```
 0 --       90772 -- 0.338467
 1 --      112126 -- 0.418090
 2 --       37420 -- 0.139530
 3 --       20412 -- 0.076111
 4 --        5142 -- 0.019173
 5 --        1247 -- 0.004650
 6 --         517 -- 0.001928
 7 --         226 -- 0.000843
 8 --         114 -- 0.000425
 9 --         170 -- 0.000634
10 --           4 -- 0.000015
11 --           4 -- 0.000015
12 --           8 -- 0.000030
13 --          16 -- 0.000060
14 --           2 -- 0.000007
15 --           0 -- 0.000000
16 --           6 -- 0.000022
```

Unless the compiler knows the parallel feature and makes use of it, ALUs exceeding 4 are almostly idle. The evaluation of instructions fetched is an iteration participated by every instruction. It is a critical path increased by the number. Fortunately, if this number is 3 or 4, the core could reach the maximum of DMIPS/MHz, and the critical path is acceptable (3/4 will get 20 ns on Altera DE2-115 FPGA, the worst condition, Only RV32I).

# Structure



"FETCH", "QUEUE","EXEC" are vectors of instructions, which have user-set length: "FTECH_LEN", "QUEUE_LEN" and "EXEC_LEN". Every one of "EXEC" has a dedicated ALU. The number of "ALU" is also "EXEC_LEN".

Instructions are fetched from "Instruction mem" and stored into "Prefetch Buffer". "FETCH" always gets instructions from "Prefetch Buffer" and has a vector of instructions from that.

In the first cycle, every one of "FETCH" will have an estimation whether it should be executed. If yes, this instruction goes into "EXEC", or into "QUEUE". In the second cycle, not-empty "QUEUE" will participate in the estimation of going to "EXEC" or "QUEUE" firstly, then "FETCH" will do that again.

The ID stage is an iteration of dispatching "QUEUE"+"FETCH" to get "QUEUE"+"EXEC". In the next stage: EX, every instruction of "EXEC" will guide its ALU to fetch Rs and write Rd to "Reg File", or send memory-operation to "LSU".

Super-scalar: "EXEC_LEN" number of instructions will be executed in the same cycle.

Out-of-order: Instructions that is not suitable in this cycle will be kept in “QUEUE" and the following instruction will have chance to be estimated and executed in this cycle. In the next cycle, data dependency changes and instructions in "QUEUE" have more possibility to be executed.

That is the basic structure of implementing super-scalar and out-of-order in this RV32IMC core.

# Register lists

Every instruction should be checked to make sure it could be executed simultaneously with others. Since an instruction can be treated as a function with inputs: Rs and outputs: Rd, this instruction could have two register lists: one for Rs, the other for Rd. If its Rs/Rd is matched with the Rs/Rd register list, this instruction couldn't be dispatched in this cycle.

One element of Rs/Rd register lists is used to tag some register that should not be utilized as a source (data stored is not updated), or written as a destination (data stored should be kept). One instruction which misses Rs/Rd register lists is issued to one of "EXEC" and sent to one of "ALU". One which matches Rs/Rd register lists is sent to one of "QUEUE", or kill the estimation of instructions if "QUEUE" is full.

Most instructions are one-cycle-effective, one of which utilizes its source and writes its destination in one cycle. There are two different kinds of instructions which will take more than one cycle: 1, instructions loading data from memory; 2, mul/div instructions.

The first kind instruction depends on memory types and CPU will have to wait for undefined number of cycles. The second kind instruction takes multiple number of cycles, which lies on implementation method.

Let's estimate the vector of instructions in "FETCH". For the first instruction of "FETCH", its Rs/Rd register list consist of Rds of LD and MUL/DIV instructions, which have long effect when they are not retired. If Rs/Rd of the first instruction of "FETCH" matches its register lists, this instruction will be sent to "QUEUE", or to "EXEC".

The second instruction of "FETCH" will inherit the Rs/Rd register list; however, CPU will add new element from the first instruction. If the first instruction is sent to "EXEC", the first instruction will offer its Rd to Rs/Rd register lists, because its Rd is on the way to the register files and fetching this register leads to error data. If the first instruction is sent to "QUEUE", its Rd is added to Rd register list and its Rs and Rd are added to Rs register list. Instructions in "QUEUE" are not executed and will be executed in the next cycle. Its Rs and Rd should be protected and any following instructions which will contaminate Rs, Rd of "QUEUE" instructions, will have to be rejected to be dispatched to "EXEC". According to this example of forming Rs/Rd register lists, every instruction of "FETCH" will have its own Rs/Rd register lists.

# Instructions

We already know the iteration of dispatching two vectors of “QUEUE","FETCH" to get another two vectors of "QUEUE", "EXEC". This processing only involves with two common kinds of instructions: LD/ST memory instructions and register-to-register ALU instructions. The excluded instructions such as jump, CSR, system, mul/div instructions will not go into "QUEUE". These instructions will be sent from "FETCH" to "EXEC" directly.

Let's discuss them below:

- FENCEI, FENCE, SYSTEM, CSR-related

  SYSTEM instructions include BREAK, SYSTEM, which will relate to CSR and lead to a jump of PC. CSR-related instructions will only read or change CSR and have no jump

operations. These instructions are rare and it is no need to put them in "QUEUE". CPU will hold them in "FETCH" until their executing condition is satisfied.

CSR-related instructions will have Rs to read. No matter how many ALUs except the last ALU are allocated, the CSR-related instruction is assigned to the last ALU. The module "SYS_CSR" will get the same Rs as the last ALU and send Rd to the last ALU. The last ALU will write data from "SYS_CSR" to the register file. The last ALU shares its resource with "SYS_CSR".

- direct jump, conditional jump instructions

  As we know, instructions following a jump instruction are not allowed to be executed. It is a wise idea to assign these jump instructions to the last ALU. No matter how many ALUs except the last ALU are allocated, jump instruction goes into the last ALU and terminates the estimation of instructions of "FETCH".

  If Rs of a jump instruction matches its register list, this jump instruction will not be executed, which will lead to freezing of "FETCH". Until it doesn't match, the jump instruction is assigned to the last ALU.

- MUL, DIV

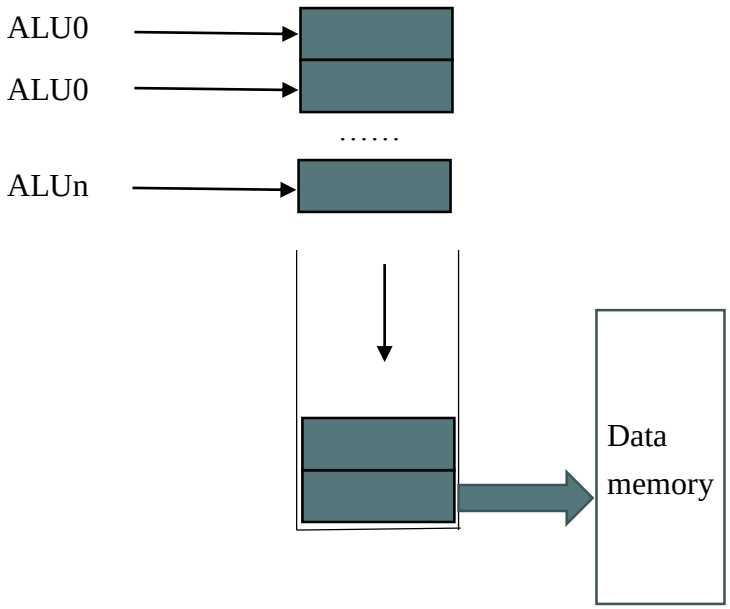
  These two kinds of instructions will take undefined number of cycles. It could be assigned to the last ALU when the MUL/DIV instruction satisfies the two conditions: 1, Rs and Rd are not ones of Rs/Rd register lists; 2, "MUL_DIV" doesn't already have one MUL/DIV instruction running.

  The last ALU will fetch Rs of “MUL_DIV" module. But its Rd data is not written through the last ALU. The Rd of MUL/DIV is available in the register lists, which prevents instructions related to this Rd being executed, but instructions un-related could be executed normally. The Rd of MUL/DIV will be written to the register file through the same channel as data from memory. In case of exception of memory instruction, the writing to register file will occur when CNT attribute of MUL/DIV instruction is decreased to 0.

# LSU

As other CPU core, SSRV CPU core has one 32-bit DATA BUS to interface with data memory. To avoid memory problem, LD/ST instructions are kept in-order. In another word, when one LD/ST instruction enters "QUEUE", no following LD/ST instructions are allowed to be executed.

Every ALU could send its memory operation command into "LSU" module. "LSU" will collect all memory operation commands into a buffer and issue one memory operation command to interface of DATA bus. Until this memory operation command retires, the next memory operation command are issued to begin its memory operation. LSU will inform other modules that it has released one memory operation.



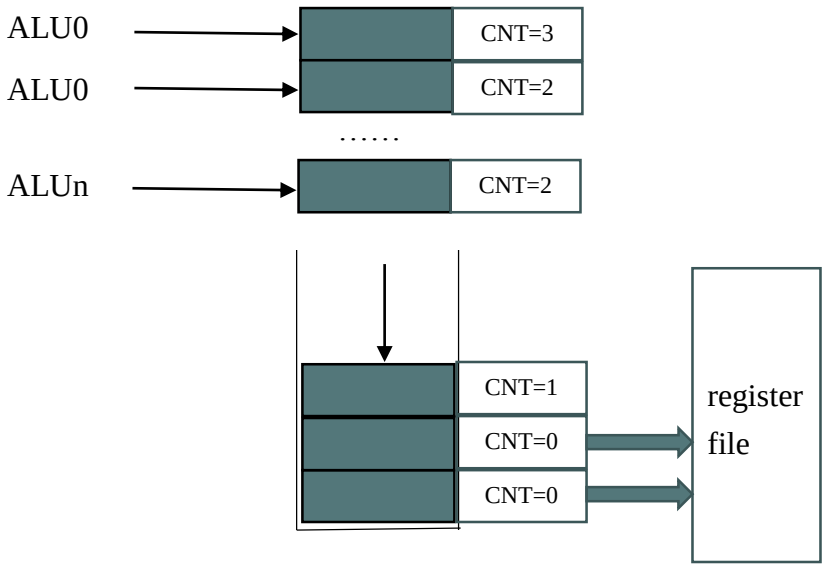

If this memory operation buffer is full, no LD/ST instruction is dispatched to any ALU.

# Register file

Each ALU could process one register-to-register ALU instruction. For precise exception service, Rds of out-of-order ALU instructions are not written to register file, but stored in a buffer. Each Rd has an attribute "CNT", which means how many memory operations are stored in memory buffer of "LSU" before it is being dispatched. When one LD/ST instruction in "LSU" is retired, which is informed by "LSU", every "CNT" is decreased 1 until it is zero. Only when "CNT" is zero, this Rd could be allowed to be written into register file and the buffer gets one free element.

If one memory operation occurs one exception, the Rds, whose "CNT" is not zero, are generated by ALU instructions which follows this exception LD/ST instruction. To flush Rds whose "CNT" is not zero will make sure that exception service program will get a register file which is not invaded by following instructions.

If the buffer for Rds is full, no more ALU instruction is dispatched to any ALU module.




# Interrupts and exceptions

Multiple instructions are dispatched out-of-order. There are 3 buffers to store different instructions:

- "QUEUE": LD/ST or ALU instructions; its format is complete; waiting for extinction of data dependency.
- memory buffer of "LSU": LD/ST instructions; its format is disappeared; only address, write data, Rd info exist; waiting for memory operation of DATA bus one by one.
- Rd buffer of "register file": ALU instructions; its format is disappeared; only the final Rd exists; waiting for "CNT" being decreased to 0 and entering register file.

When an interrupt or exception occurs, how to deal with instructions stored in 3 buffers?

If an interrupt occurs, the direction of program will continue when this interrupt is finished. It is not necessary to have a precise breakpoint because it will continue dispatching instructions when the interrupt service entry is over. There is one mode to deal with this condition: DIRECT MODE.

In this mode, 3 buffers are frozen and new instructions from interrupt service entry are executed in-order strictly. Only the last ALU is valid and no new instruction is dispatched until the last one is retired. When the interrupt service is over and leaves DIRECT MODE, CPU keeps "FETCH" the same as before and 3 buffers are unfrozen. CPU will continue its out-of-order dispatching.

As for exceptions, there are 2 types of exceptions, which need precise breakpoint. One is LD/ST memory-operation instruction. When a LD/ST instruction is started in the DATA bus, the register file will keep not contaminated by the following instruction. If data memory reports error, CPU will assert that the breakpoint is the very position of this exception-generated LD/ST instruction.

The other is divide-zero exception of DIV instructions. If this is needed, CPU makes sure this DIV instruction will be dispatched in-order strictly. If divide-zero is detected, this exception will have a precise breakpoint and in that cycle, no following instruction is issued .

# Summary

SSRV CPU core will make most use of RISC advantage. LD/ST instructions and register-to-register ALU instructions are the most of all instructions. Optimization of them is the key of improving performance. This core will dispatch multiple instructions in one cycle and make sure writing to register files until all instructions retire, especially LD/ST instructions.

The parameters mentioned above could be configured in one file. I have finished most of design and make sure it could run programs smoothly . But it is not an entire CPU core. It lacks of clear CSR support, interrupts and exceptions.