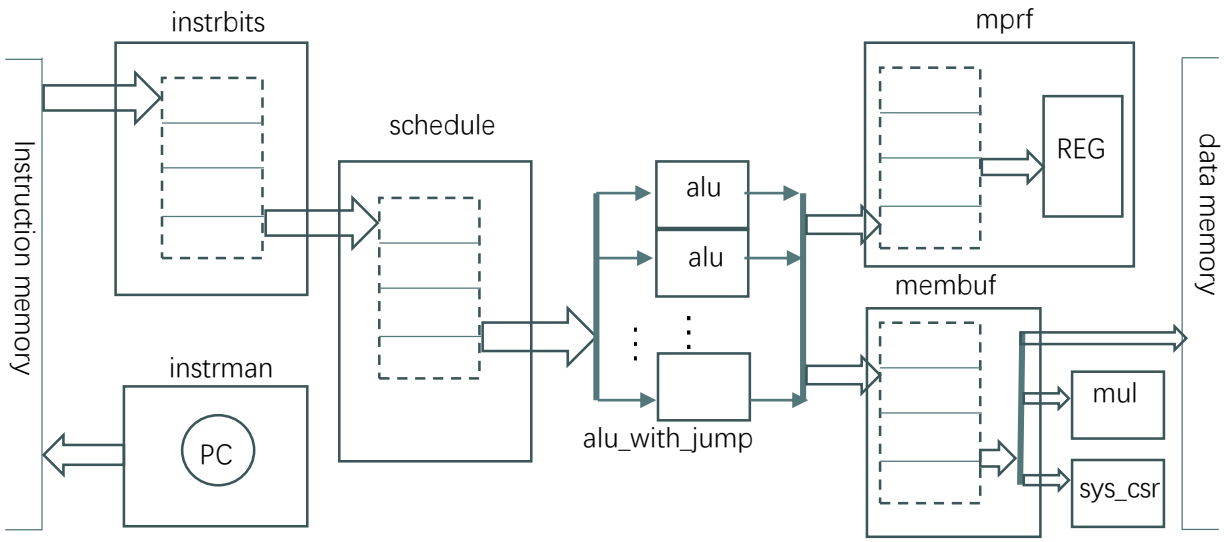
SSRV tutorial

Overview

The goal of this document is to describe how SSRV implements out-of-order and super-scalar. SSRV is a 3-stage RV32IMC CPU core. Different from rivals, SSRV is configurable to adjust levels of out-of-order and super-scalar via 3 parameters. Besides these 3 ones, there are more parameters to effect performance.

SSRV is based on 4 different multiple-in, multiple-out buffers connected with each other. The central of them is built in “schedule” module, which has “FETCH_LEN” inputs, “EXEC_LEN” outputs and a capacity of “SDBUF_LEN” instructions.



If these 3 parameters are given different values, this core will show different Dhrystone Benchmark scores. The next table will list how these key parameters produces different performance cores.

FETCH_LEN--SDBUF_LEN--EXEC_LEN	DHRY(best)	DMIPS/MHz(best)	DHRY(legal)	DMIPS/MHz(legal)
1—1—1	5205	2.96	2645	1.51
1—2—1	5205	2.96	2659	1.51
2—2—2	6366	3.62	3344	1.90
2—3—2	6407	3.65	3471	1.98
2—4—2	6407	3.65	3520	2.00
2—6—2	6407	3.65	3533	2.01
3—3—3	6708	3.82	3689	2.10
3—4—3	6753	3.84	3758	2.14
3—6—3	6799	3.87	3787	2.16
4—4—4	6893	3.92	3758	2.14

4—5—4	6941	3.95	3801	2.16
4—6—4	6941	3.95	3816	2.17
8—16—8	7038	4.01	3906	2.22
16—32—16	7038	4.01	3921	2.23

“EXEC_LEN” is a parameter of super-scalar, which determines how many ALUs are instantiated to execute instructions in the same cycle. “SDBUF_LEN” is a parameter of out-of-order, which means how many instructions are evaluated to present “EXEC_LEN” instructions, the bigger it is, the more possibility to stuff ALUs.

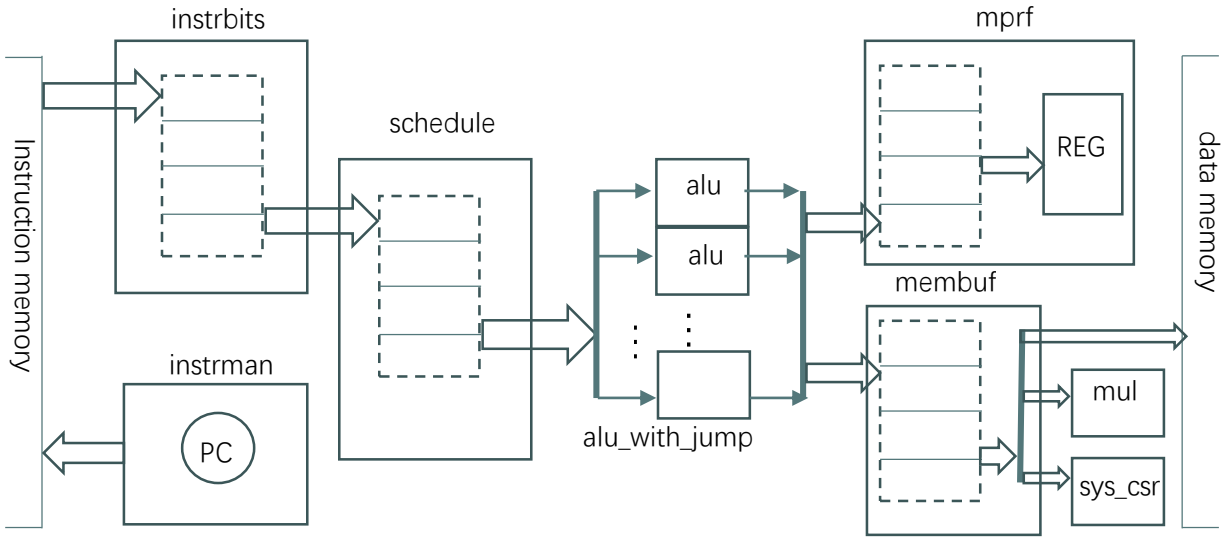
More than that, these 3 parameters can be random integer. There is a status report when all are assigned to 16. It is obvious that SSRV is a robust solution of out-of-order and super scalar.

```
ticks =      261273  instructions =      282644  I/T = 1.081796
          NUM          TICKS       RATIO
            0 --       87638 -- 0.335427
            1 --      108594 -- 0.415634
            2 --       37830 -- 0.144791
            3 --       19693 -- 0.075373
            4 --        3562 -- 0.013633
            5 --        1793 -- 0.006863
            6 --         949 -- 0.003632
            7 --         566 -- 0.002166
            8 --         110 -- 0.000421
            9 --          58 -- 0.000222
           10 --         360 -- 0.001378
           11 --          88 -- 0.000337
           12 --           0 -- 0.000000
           13 --          12 -- 0.000046
           14 --           0 -- 0.000000
           15 --           4 -- 0.000015
           16 --          16 -- 0.000061
```

All files of SSRV are synthesizable and aimed to provide a high-performance core for ASIC and FPGA. Except for a file “sys_csr.v”, which is related to interrupt/exception and system control, others could be unmodified to be instantiated as sub-modules.

If you want to utilize SSRV to build a high-performance CPU core of your own, just modify “sys_csr.v” to have your own system control solution and combine that with other files to be your high-performance core. You are free to choose appropriate parameters, which will give your balance between performance and logic cell cost.

An example on how instructions are managed



Let's take an example to discuss how hinstructions are managed.

```
138e:    01059313            slli   t1,a1,0x10
1392:    01035f93            srli   t6,t1,0x10
1396:    01efc333            xor    t1,t6,t5
139a:    030e                       slli   t1,t1,0x3
139c:    07837313            andi   t1,t1,120
13a0:    007fff93            andi   t6,t6,7
13a4:    c398                       sw     a4,0(a5)
13a6:    01f36733            or     a4,t1,t6
13aa:    c11c                       sw     a5,0(a0)
13ac:    00871313            slli   t1,a4,0x8
13b0:    c3d4                       sw     a3,4(a5)
13b2:    00e36fb3            or     t6,t1,a4
13b6:    01f69023            sh     t6,0(a3)
13ba:    01d69123            sh     t4,2(a3)
13be:    873e                       mv     a4,a5
13c0:    869e                       mv     a3,t2
13c2:    8796                       mv     a5,t0
```

In the first stage, "instrman" module will initiate a request to "instruction memory", which has a PC address : 13b0. Since it could fetch 4*32 bits, hex data : "873e_01d69123_01f69023_00e36fb3_c3d4" will appear on the imem bus "imm_rdata" in the next cycle, which is the instructions of addresses: "13be+13ba+13b6+13b2+13b0".

In the second stage, a multiple-in, multiple-out buffer in the module "instrbits" will welcome its incoming data: imem_rdata "873e_01d69123_01f69023_00e36fb3_c3d4". This buffer has stored two instructions: "00871313_c11c", whose addresses are 13ac and 13aa. These two clusters of hex data will join together: 873e_01d69123_01f69023_00e36fb3_c3d4_00871313_c11c. From the

lowest bit, "FETCH_LEN",which is 4 here, instructions will be gathered to output to the next module. Now, we know the addresses of the 4 instructions are 13b2, 13b0, 13ac, 13aa.

The multple-in, multiple-out buffer of the module "schedule" has 4 incoming new instrctions: 13b2, 13b0, 13ac, 13aa. Since the buffer has stored 3 instructions: 13a6, 139c, 139a and its capacity is "SDBUF_LEN", which is 6 here, it will have to accept 3 of 4.

These 6 instructions are candidates:

```
139a:    030e                    slli   t1,t1,0x3
139c:    07837313          andi   t1,t1,120
13a6:    01f36733          or     a4,t1,t6
13aa:    c11c                    sw     a5,0(a0)
13ac:    00871313          slli   t1,a4,0x8
13b0:    c3d4                    sw     a3,4(a5)
```

We will choose "EXEC_LEN",which is 4 here, instructions between them. Since these is always data dependency of instructions, the instructions : 139a, 13aa and 13b0 are winers.The losers will stay in the buffer to wait for the next try. These 3 winers are going to registers, which means the end of the second stage.

```
139a:    030e                    slli   t1,t1,0x3
13aa:    c11c                    sw     a5,0(a0)
13b0:    c3d4                    sw     a3,4(a5)
```

In the third stage,these 3 instructions will pass through their own ALU module. The instruction 139a will have a destination register: t1 and its new data. The othes are different because they need operations on data bus, not just a destination register and a simple new data.

The buffer in the module "membuf" will have two newcomers: 13aa and 13b0, which are queued to operate the data bus one by one. The buffer have accepted one instrction : 13a4 in the last cycle. It is operating the data bus now. The newcomers have to be queued in the buffer.

```
13a4:    c398                    sw     a4,0(a5)
```

The instruction 139a will have a different destination: mprf module. The inner buffer will accommodate it and make them be queued after two early instructions: 13a0 and 1396.

```
1396:    01efc333          xor    t1,t6,t5
13a0:    007fff93          andi   t6,t6,7
```

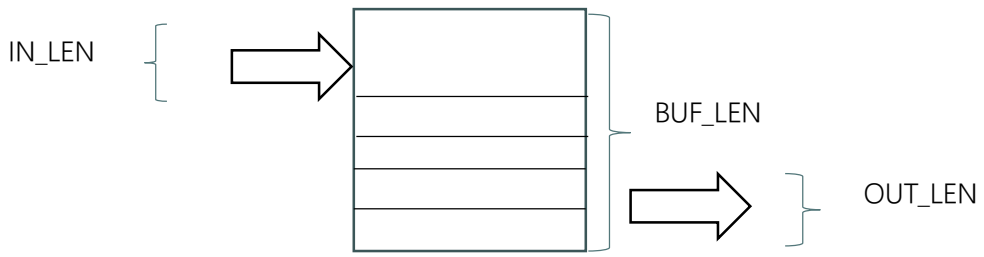
These 3 instructions will not be allowed to write to the register file: R1~R31 directly. The instruction 13a4 are operating the data bus now, and we do not know whether it will be successful.

If it fails, instructions following it will cause mess when an exception occurs. There must be a choose from the buffer, and only instuctions ahead of the instruction: 13a4 have possiblity to write to the register file. The instructions following it will have to wait for the acknowlegement of data bus in the buffer. After that, they will be allowed to write to the register file.

Multiple-in, Multiple-out Buffer

The basic component of SSRV is a multiple-in, multiple-out buffer, which has a capacity of multiple elements.

It has three parameters: IN_LEN, which defines how many elements are coming in from outside; BUF_LEN, which define the maximum elements could be accommodated; OUT_LEN, which define how many elements are going out.



Different parameters lead to different capabilities of the buffer. Because of unpredictable condition on processing instructions, this kind of buffer is very helpful in keeping instructions.

FIFO is a common buffer of this, which is always 1-in, 1-out. How to define a multiple-in, multiple-out FIFO? It assumes that “in_data” are “IN_LEN” numbers of “XLEN”-bit data, “out_data” are “OUT_LEN” numbers of “XLEN”-bit data. This FIFO is “buf_data”, which has “BUF_LEN” numbers of “XLEN” bits. We can have Verilog statements like this:

```
assign temp_data = buf_data|(in_data>>(buf_length*XLEN);

assign out_data = temp_data;
```

“temp_data” is composed of “buf_data” and “in_data”. Its length is “buf_length” + “in_length”. However, when “temp_data” is to stored to “buf_data”, its length should be “buf_length” + “in_length” – “out_length”, and data should be “ temp_data>>(out_length*XLEN)” , because of outputting “out_data”.

These 3 parameters define the maximum number. IN_LEN means there could perhaps be 0,1,to IN_LEN number of elements coming to the buffer. It has an array of the length: IN_LEN. If there are 2 elements, No.0 and No.1 are valid ones and others are filled with nothing. If the input is full, No.0, No.1, to No.(IN_LEN-1) are all valid.

Here is some definitions of input arrays.

```
wire `N(IN_LEN)       in_vld;
wire `N(IN_LEN*XLEN)  in_instr;
wire `N(IN_LEN*XLEN)  in_pc;
```

“N(n)” is a Verilog definition: `define N(n) [(n)-1:0].

If in_vld is the form like this: ‘b1, ‘b111, ‘b1111, it is easy to combine it with the buffer like this:

```
assign temp_instr = buf_instr|(in_instr<<(buf_length*XLEN);
```

If in_vld is like this: ‘b101, 1’b11010, the empty “instr” or “pc” should be removed from arrays because the buffer will not contain empty ones. Here is the method to remove empty elements and form a new array without empty ones.

```
wire `N(IN_OFF)           cnt_num `N(IN_LEN+1);
wire `N(IN_LEN)         chain_vld `N(IN_LEN+1);
wire `N(IN_LEN*XLEN)  chain_instr `N(IN_LEN+1);
wire `N(IN_LEN*XLEN)     chain_pc `N(IN_LEN+1);

assign     cnt_num[0] = 0;
assign   chain_vld[0] = 0;
assign chain_instr[0] = 0;
assign    chain_pc[0] = 0;

generate
genvar i;
for (i=0;i<IN_LEN;i=i+1) begin
    wire              vld = in_vld[i];
    wire `N(XLEN)   instr = in_instr>>(i*XLEN);
    wire `N(XLEN)      pc = in_pc>>(i*XLEN);
    wire `N(IN_OFF) shift = vld ? cnt_num[i] : IN_LEN;

    assign cnt_num[i+1]     = cnt_num[i] + vld;
```

```
    assign chain_vld[i+1]   = chain_vld[i]|(vld<<shift);

    assign chain_instr[i+1] = chain_instr[i]|(instr<<(shift*XLEN));

    assign chain_pc[i+1]    = chain_pc[i]|(pc<<(shift*XLEN));

end
endgenerate
```

Eventually, there are new arrays: chain_vld[IN_LEN], chain_instr[IN_LEN], chain_pc[IN_LEN], which are the result of removing empty elements. We use a standard: in_vld[i]==1 to screen original arrays. Elements of “in_vld==0” is abandoned by getting a shift: IN_LEN, which makes the empty element beyond the boundary of the array. Anyway, if elements of "in_vld==0" need be kept, we can have another "cnt_num[i]" to give each of them a shift number and they will gather to be a new array.

The Verilog style of describing the multiple-in, multiple-out buffer here will lead us undstanding how to fliter an array or split this array into two different arrays. It is a necessary method to manage an array.

Let's consider how to connect two buffers. Their connections can have 3 styles.

- Orphan mode

  When the slave is sure that it could contain the maximum of incoming, the slave give a signal of fetching to the master. No matter how many is coming actually, there is no problem of overflow. The slave is initiative and the master does not need to know how many elements are kept in the slave.

- Mother mode

  The master is initiative and it will always give the slave elements as many as possible. The slave should answer how many are acceptted acutally. The master will abandon them and make sure that it will send the closest elements in the next cycle. There is an interaction between them.

- Father mode

The master is aware how many elements are needed by the slave. It will send adequate number of elements to the slave. There is no interaction and the slave just accepts its coming elements. The slave need not worry about overflow.

These 3 styles are all used in SSRV.

The buffer of "instrbits" will store instructions from instruction memory. It fetches instructions when the empty space of this buffer is enough. Its connections to the instruction memory is a style of "orphan" mode.

The buffer of "instrbits" will provide instructions to the buffer of "schedule". The module "schedule" will try to issue instructions as much as it can, but it will not be sure how many are issued successfully. It needs the buffer of "instrbits" gives the maximum number of instructions, and it will return how many are issued to the next stage or kept in its buffer. The connections between them are "mother" mode.

The module "schedule" is aware of the empty space of both the buffers of "membuf" and "mprf". They are necessory to help it issue how many "OP/OP-IMM" instucutions or "MEM" instructions. It will make sure that its schedule is satisfied by all its slaves. The buffers of "membuf" and "mprf" need not consider the problem of overflow and it is a problem of the module "schedule". Their connections are "father" mode.

Instructions

SSRV has two buffers to handle two different kinds of instructions. One is for alu instructions in the "mprf" module, which are OP or OP-IMM instructions of RV32IC. The other is for mem instructions in the "membuf" module, which are ones related with memory operation.

It is not possible to retire two mem instructions when there is only one data bus. The buffer of "membuf" module gathers some mem instructions and issue one of them in every cycle. Each of them is possible to cause an exception when data bus reports error. The mem instructions could be called "major" instructions.

The alu instructions could be called "minor" instructions. When they have a chance to be executed, their destination register numbers and updating data are queued in the buffer, until their previous mem instructions are all retired. It is necessary to make sure no invasion to the register file from its following alu instructions when an exception occurs.

	alu0	alu1	mem0	alu2	alu3	mem1	alu4
order	0	0	1	1	1	2	2

To distinguish alu instructions, there is a method that is to give every instructions one order. When the current instruction is one of mem instructions, its order will plus one, or keep the same. The first mem instructions of the "membuf" buffer will always have an order: 1, which is the current instruction operating the data bus. Instructions, which have the order 0, will be allowed to update the register file. If the current mem instruction is reporting successful, the order of each alu instruction will decrease one until it reaches 0. That a mem instruction retires successfully will bring more alu instructions with "0" order.

Every cycle, SSRV tries to retire one mem instruction and multiple alu instructions. According to their empty space of both buffers, "schedule" module will issue indefinite instructions and try to fill these two buffers.

So, there are two basic kinds of instructions:

- alu: only related with the registers:R1~R31, includes: OP/OP-IMM of RV32IC
- mem: load and store instructions

However, there are two kinds of instructions, which will have to be dealed with as the same as mem instructions.They are:

- mul: multiply-divide instructions
- csr: exchange data between CSR and general-purpose registers.

The mul instructions wll have multiple cycle to be executed. Each mem instruction will also need varied cycles to complete. In case of that, the mul instruction can be treated as one special memory-loading instruction.

The csr instruction will exchange one general-purpose register with one CSR register. It could be treated as one special mem instruction: loading and storing in the same cycle.

These two kinds of instructions will have their own exclusive cycle to complete operation. Below, MEM means three kind of instructions: mem, mul and csr, which are dedicated to the “membuf” module. ALU means alu instructions, whose operation result enters the “mprf” module.

MEM and ALU instructions can be scheduled out-of-order, which means its following instructions could be issued before it does. Instructions introduced next are not allowed because its following ones are disabled permanently or temporarily.

- err : error occurs when fetching this instruction, treated as a special instruction.
- illegal: fetching successfully, but it does not belong any defined RV32 instructions.
- fencei: one kind of RV32I instructions
- fence : one kind of RV32I instructions.
- sys : system instructions, often related to CSR, such as: break, call.
- jalr : jump instructions, target address is provided by one general-purpose register.
- jal : jump instructions, target address is related to PC.
- jcond: conditional jump instructions.

These 8 kinds of instructions are named “SPECIAL” instructions.

When the “schedule” module initializes a scheduling process between “SDBUF_LEN” number of instructions, SPECIAL instructions should be treated as the last one because its following ones cannot be executed unless they are retired. Also, it is not possible that two of SPECIAL instructions exist in the same list of the “schedule” module.

SPECIAL instructions can have two processing styles just like MEM and ALU ones do.

“jalr, jal, jcond” are ALU-like SPECIAL instructions. One of them could change PC to its target address and make instructions of the target address queued after instructions ahead of this jump instruction. It can trigger jump operation before some MEM instructions ahead of it are queued.

The others are MEM-like SPECIAL instructions. One of them will wait until all MEM instructions ahead of it are retired successfully. It involves with changing some CSRs and it is not revoked. So, it should be treated as one of MEM instruction.

Architecture of SSRV