Contents

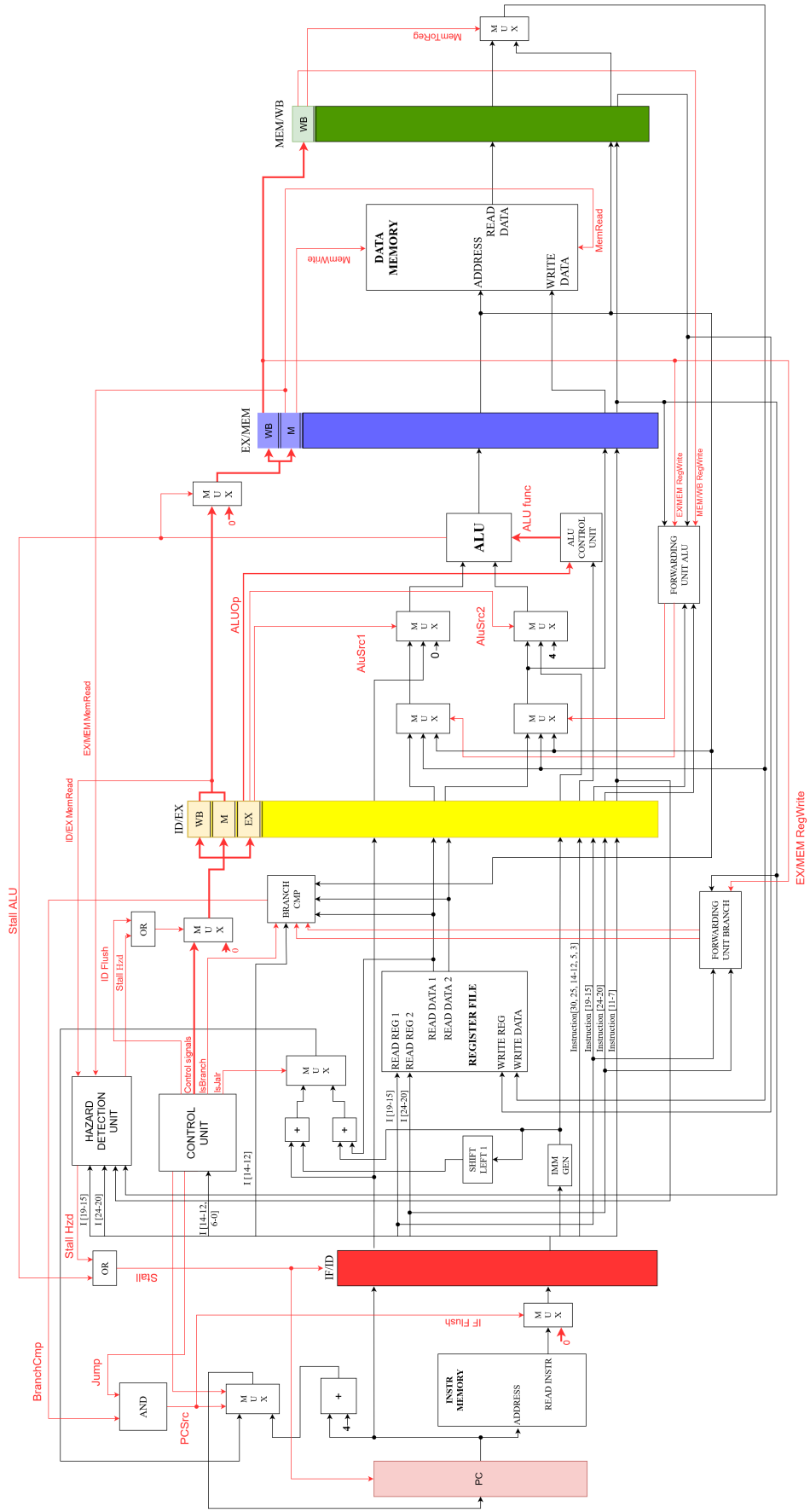

Figure 1: Pipeline Schematic

Abstract

In this project for didactic purposes a 5-stage pipeline of Risc-V has been implemented, described entirely in vhdl. The implemented instruction set architecture (ISA) is that of RV64IM [1], therefore multiply-divide-remainder instructions have been implemented alongside all basic 64-bit integer instructions, with the exception of CSR instructions and FENCE, FENCE.I instructions. Unimplemented instructions are executed as NOP. The pipeline was tested with instructions extracted from a simple bubble sort program and and an arithmetic operations program.
In this document a general description of the designed hardware will be given, then a description of the test programs used and a show of the results obtained.

1 Hardware Description

A five stage pipeline was implemented: Instruction Fetching, Instruction Decoding, Execution, Memory reading/writing, Writeback. In Figure 1 the schematic of the pipeline is reported. The data and the control lines are stored and propagated in the registers contained in the divisions between stages. The datapath includes an Instruction Memory and a Data Memory, that are single port and read-write in a single clock cycle, and a Register File. Forwarding logic is implemented and there is a Branch Comparison Unit early in the ID stage. Forwarding units are needed both for the ALU and the Branch Comparison Unit. A Control Unit analyzes the instruction and generates the control signals for all the stages of the pipeline, while the Hazard Detection Unit manages data hazards.
What follows is a more detailed description of some of the components:

PC
A simple register that is updated only when the *Stall* control signal coming from the Hazard Detection Unit or the ALU is not zero. Like all the other registers, except the ones in the Register File, it is updated on the clock falling edge. At every simulation start it is set to zero.

Instruction Memory
The Instruction Memory is a read only memory designed in a primitive way for the sake of this project's simple simulations. It's content is a fixed array of 32-bit elements that represent the set of encoded RV64IM instructions that the pipeline should execute. The address to the array is given by the PC signal, precisely the PC register output divided by four (least significant two bits ignored). PC=8 and hypothetically PC=11 will address the same location, but it is assumed anyway that in all operated simulations the PC will only change in steps of four (bytes) starting from zero.
So the instructions had to be inserted by hand before executing a simulation. The procedure followed was taking the given program written in C, obtaining the assembly code and then using the *objdump* command to decode the assembly and obtain the instructions written in hex. The jump and branch type instructions had values for the jump amount in bytes that didn't match with the Instruction Memory array indexing setup so they had to be adjusted by hand.

Register File
The Register File includes thirty-two 64 bit registers. The writing happens on the rising edge of the clock desynchronized from the other registers of the pipeline, that update on falling edge. This way, the data that enters the MEM/WB division at the beginning of a pipeline cycle, on falling edge, can be written to the Register File in the same cycle, on rising edge. Moreover the Register File reading

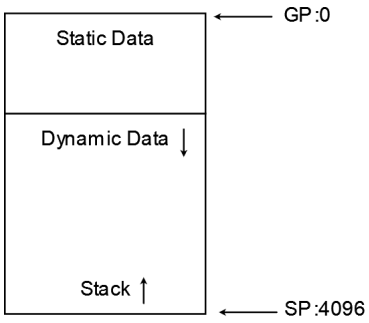

Figure 2: Data memory

is done asynchronously and so writing and reading happen in the same pipeline cycle.
Writing only really happens if the control line *RegWrite* is asserted and, if the destination register is the RiscV register x0, what's written is always zero, independently from what the *WRITE DATA* bus contains. Default values of x2(*sp*) and x8(*s0/fp*) registers are set to point to the bottom of the Data Memory and x3(*gp*) is set to point somewhere to the middle (see Data Memory paragraph). These are the values of *sp*, *s0/fp* and *gp* when a simulation starts.

Data Memory
The Data Memory is written on the falling edge of the clock and read asynchronously. It is large 4096 bytes and divided in a static memory segment and a a data memory segment, see figure 2. As mentioned above the default value of the *gp* points to the bottom of the static segment, the *sp* and *s0/fp* to the bottom of the data segment. The 4096 bytes are structured in banks of 8 bytes for a total of 512 banks. This is done to allow the functioning of instructions that work on data of type word, half-word or byte. A Store Word SW instruction for example, can store data in the first four or in the second four bytes of a bank. Another placement is not acceptable and causes a misalignment error. A load Halfword instruction will read only from bytes 0-1 , 2-3 , 4-5 , 5-6 or 7-8, not from bytes 1-2 or 3-4 or so on.
In the actual vhdl code an interface (*datamem_interface.vhd*) handles the addressing and loading and storing in and from the memory divided in banks. Inside it the eight 1-byte blocks that constitute a bank are instantiated. A separated entity *datamem* (in *datamem.vhd*) describes one such 8-bit block. The interface receives as input the address in bytes contained in an instruction, it divides it by eight to obtain the address of a bank and uses it's least significant three bits to know which of the eight 1-byte-blocks need to be accessed. The other input it receives is the instruction *func3* code that tells it whether the instruction is a double, word, halfword or byte operation. According to this, plus the three-bits offset mentioned above, it decides which of the eight 1-byte blocks of the bank need to be loaded or read from.

Control Unit and Immediate Generator
The Control Unit simply sets the control lines according to the instruction's opcode field. As its customary, to avoid having to deploy all the control logic in a single unit, a two-bit *ALUOp* signal is set as output. It will be used by the ALU Control Unit to determine the operation needed from the ALU. If an invalid opcode is provided the control lines are all set to zero to work as a NOP.

Meanwhile the Immediate Generator unit searches among the instruction bits to produce the right immediate, according to the instruction type encoded in the opcode and funct3 fields [1].

Branch Comparison Unit
The Branch Comparison Unit is placed in the ID stage to determine the realization of the branch condition in branch instructions (BEQ, BGE, etc.) independently from the ALU and earlier than at EX stage.
This is done to minimize pipeline cycles lost if a branch is taken.
The adopted strategy is a simple static prediction that a branch is not taken. The instruction following a branch instruction is loaded in the pipeline to continue normal operation, but it is flushed before entering IF/ID if the branch is determined taken (see *IF_FLush* in the schematic Fig. 1 [1]).

Forwarding Logic
Two Forwarding units are needed, one for the ALU and one for the Branch Comparison Unit. They simply implement the logic conditions (the *.vhd* files can be consulted to examine the conditions).

Hazard Detection Unit
This is a standard hazard unit that detects hazard conditions that can't be solved by forwarding and sends a *Stall_Hzd* signal. The *Stall_Hzd* signal combines in an OR in the datapath with the *Stall_ALU* signal potentially asserted from the ALU (see next).

ALU
An ALU Control Unit receives the *ALUOp* signal, the func3 field and a few other bits of the instruction and outputs an *ALU_func* signal to tell the ALU which operation to execute. The ALU includes a 64-bit adder-subtractor and an hardware block to to do multiplication and division. For this last part there are two configurations of the ALU, one that does multiplication and division using the same hardware block and another that uses different blocks, one for multiplication, one for division.

The first version uses a shift-add multiplier and a divider using the same registers and the same adder-subtractor. See the schemes in Fig.3. The functioning is standard. It's reported below for reference:

Multiplier:
1. If the least-significant bit of A is 1, then register B, containing b_{n-1} $b_{n-2}...b_0$ is added to P; otherwise, 00...00 is added to P.
2. The sum is placed back into P.
Registers P and A are shifted right, with the carry-out of the sum being moved into the high-order bit of P, the low-order bit of P being moved into register A, and the rightmost bit of A, which is not used in the rest of the algorithm, being shifted out.

Divider:
a/b, a in A, b in B, 0 in P. Produces one quotient bit at a time in A, remainder In P.
1. Shift the register pair (P,A) one bit left.
2. Subtract the content of register B (which is b_{n-1} $b_{n-2}...b_0$) from register P, putting the result back

[1] In this design *PCSrc* is asserted when a jump in PC is needed because of a jump or a taken branch instruction, so it is equivalent to *IF_Flush*

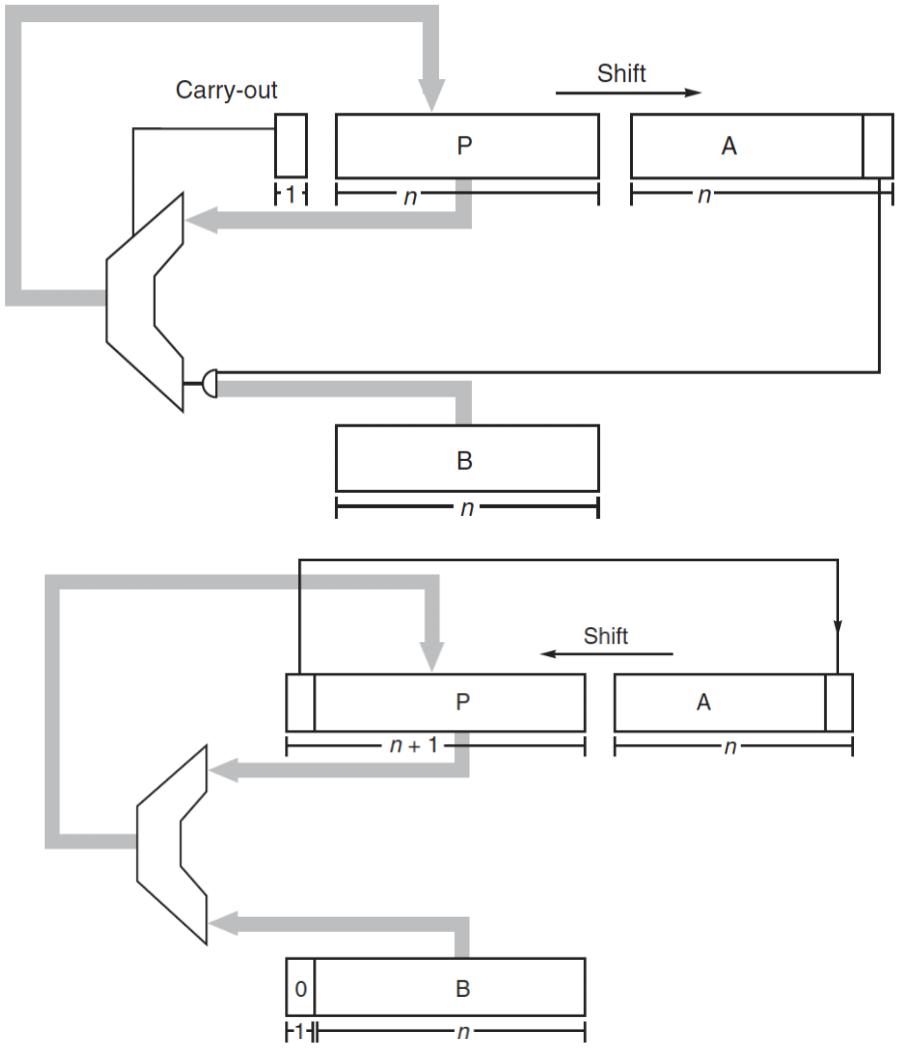

Figure 3: Schemes for shift-add multiplier above and divider below

into P.
3. If the result of step 2 is negative, set the low-order bit of A to 0, otherwise to 1.
4. If the result of step 2 is negative, restore the old value of P by adding the contents of register B back into P.

The implementation uses a single 129-bit accumulator register for 1bit+P+A. Moreover the AND shown in the scheme of the multiplier between the rightmost bit of the accumulator (of A) and B is setup to always let B through in the case of division, because in this case instead of the rightmost bit of A, a signal always equal 1 is sent.
For the additions and subtractions an adder-subtractor is used that can do sum for multiplication and subtraction for division.
The implementation of the algorithms uses the pipeline clock and the steps reported above all happen in a single clock cycle. To keep count of the cycles a counter register is used. One clock cycle is spent to initialize the registers with the data coming from the ALU. After that a *BUSY* signal is activated and it stays asserted for the duration of the multiplication/division, which lasts an additional 64 clock cycles. The *BUSY* signal is used as output of the ALU to stall the pipeline during this time (see schematic and *Stall_ALU* signal). After the 65th clock cycle the operation is completed, *BUSY* is de-asserted and the pipeline can continue with the next cycle. The multiplier-divider can operate with 32-bit operands for ALU Word operations (MULW, DIVW, etc.). In this case a *IsW* signal is set to '1' by the ALU as input to the multiplier-divider. With *IsW* asserted only half the bits of the operand registers are used, the rest are set to zero and the number of clock cycles at which the operation is stopped is 33. Similarly a *IsSigned* signal is given as input to signal the operands should be interpreted as signed. The operands' sign bits are checked and loaded in a register, negative operands are made positive with 2's complement so that they can be interpreted as unsigned and at the end of the operation the result is given the correct sign according to the initial operands' signs.

The second configuration of the ALU uses the same divider, but a different multiplier that uses Booth's algorithm. See Fig.4. The algorithm operates on the fact that strings of 0's in the multiplier require no addition but just shifting and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as 2^{k+1} to 2^m. The algorithm requires examination of the multiplier bits and shifting of the partial product and it works like this:

One clock cycle for initializing with SC at 0, putting multiplicand in BR, multiplier in QR and setting AC to 0 and the extra bit of QR, Q_{n+1} to 0. Afterwards:
1. The two bits of the multiplier in Q_n and Q_{n+1} are inspected. If the two bits are equal to 10, it means that the first 1 in a string has been encountered. This requires subtraction of the multiplicand from the partial product in AC. If the 2 bits are equal to 01, it means that the first 0 in a string of 0's has been encountered. This requires the addition of the multiplicand to the partial product in AC. When the two bits are equal, the partial product does not change. An overflow cannot occur because the addition and subtraction of the multiplicand follow each other.
2. The next step is to shift right the partial product and the multiplier (including Q_{n+1}). This is an arithmetic shift right (ashr) operation which shifts AC and QR to the right and leaves the sign bit in AC unchanged.
3. The sequence counter is incremented and the loop is repeated 64 times for a Double, 32 for a Word division.

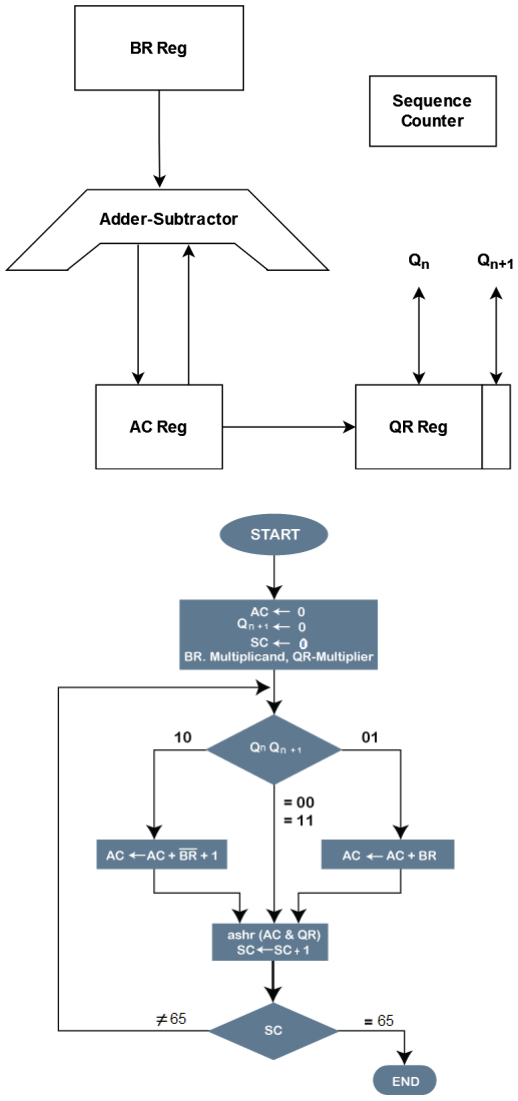

Figure 4: Hardware and flowchart for Booth's multiplier algorithm

This design has the disadvantage of requiring a realization in a different hardware block from the divider, because of a different register setup and it still employs 65 clock cycles for initialization and operation.
It has the advantage of not needing any logic for distinguishing signed from unsigned operation, since Booth's algorithm works for both interpretations of the operands.
Either way it's clear that both versions of the multiplier are slow and a faster multiplier requiring less or even a single clock cycle can be considered, for a trade off with hardware size.

Regarding the divider, the case of division by zero is handled by setting a default value of all bits to 1 for the quotient and putting the dividend in the remainder.

As output the ALU emits also a 3-bit vector to signal if there's been an arithmetic exception and of which type. This means an overflow during an ADD, or ADDW, SUB or SUBW instructions, or a division by zero during a DIV/REM, or DIVW/REMW type instruction.
The overflow flag in addition/subtraction comes from the Overflow bit emitted as output by the adder-subtractor. This is obtained from the XOR of the last and the second last carryout bits in the 64-bit adder-subtractor. It immediately signals if there's an overflow in a signed addition or subtraction (and in the RV64IM ISA ADD or SUB are always signed operations). To have the same flag for an ADDW or SUBW (32-bit) Word operation an analogous XOR was implemented between the thirty-second and thirty-first bit of the adder-subtractor used in the ALU.
The division by zero flag instead is extracted from the divider.

For what concerns shifting instead, at present the instructions of the shift family utilize the shift functions from the ieee.numeric_std library that at simulation execute shifts in one cycle. A barrel shifter could be added to the ALU to replicate structurally this behaviour.
Possible overflow flagging during shift left hasn't been considered.

Arithmetic Exception Unit
The exception flag vector extracted from the ALU is brought to a simple exception unit, that uses the clock as input to store the type of exception in a register at clock falling edge and at the same time to emit a warning message that can be read during simulation. In fact the use of the clock edge was necessary to make it possible to emit this warning message a single time during simulation when a flag is up and the register isn't really used anywhere and was only added to be available at simulation for visualization and didactic purpose.
With this implementation the exception type register is updated, and the message is emitted, one clock cycle after the exception happened in the EX stage in the ALU.

2 Testing

2.1 Bubble Sort

The first program used for testing the pipeline was a very simple bubble-sort program. The C code is reported here:

```
#### bubble_sort.c ###
#include <stdio.h>
```

```
int main()
{
 int arr[5];
 arr[0] = 3;
 arr[1] = 5;
 arr[2] = 1;
 arr[3] = 2;
 arr[4] = 4;

 int tmp;

 for(int i = 0; i < 4; i++){
     for(int j = 0; j < 4 - i; j++){
         if(arr[j] > arr[j + 1]){
             tmp = arr[j];
             arr[j] = arr[j + 1];
             arr[j + 1] = tmp;
         }
     }
 }
 return 0;
}
```

From this program, the instructions to be loaded in the Instruction Memory where obtained with the procedure previously described. They are reported in the Appendix 3.1.
The first instructions prepare the stack. As can be seen the integer numbers 3,5,1,2,4 get stored as Words from the locations -48(s0) to -32(s0), occupying four bytes each. As previously mentioned, at the beginning of simulation *sp* and *s0* are set to the bottom of the Data Memory, so, going back to the visualization of the Data Memory as made of banks of 8 bytes and remembering that it is long 4096 bytes, so 512 banks, it can be seen that the location of *s0* is the end of the last bank, *membank_511* and the location -48(s0) is six bytes above, so it is the beginning of *membank_506* (note that the first membank is called *membank_0*). The second location -44(s0) is the second half of *membank_506* (the data is 4-bytes so two elements fit in one membank), the third -40(s0) is the beginning of *membank_507* and so on.
The same locations are used to store the data at the end of the instruction list execution.
So at the end it is expected to find the numbers 3,5,1,2,4 sorted in ascending order from *membank_506* to *membank_508* 's first half. Figure 5 shows the results align with this expectation.

2.2 Arithmetic Program

While they were being implemented, the different components of the datapath were tested with the dedicated test-benches to ascertain their functioning. This applies to the ALU and the multiplier-divider as well, for the arithmetic part.
Beyond that though, a very simple C program focused on a few arithmetic operations was used for a test with the whole pipeline. This program is reported in the Appendix 3.2 and the instructions

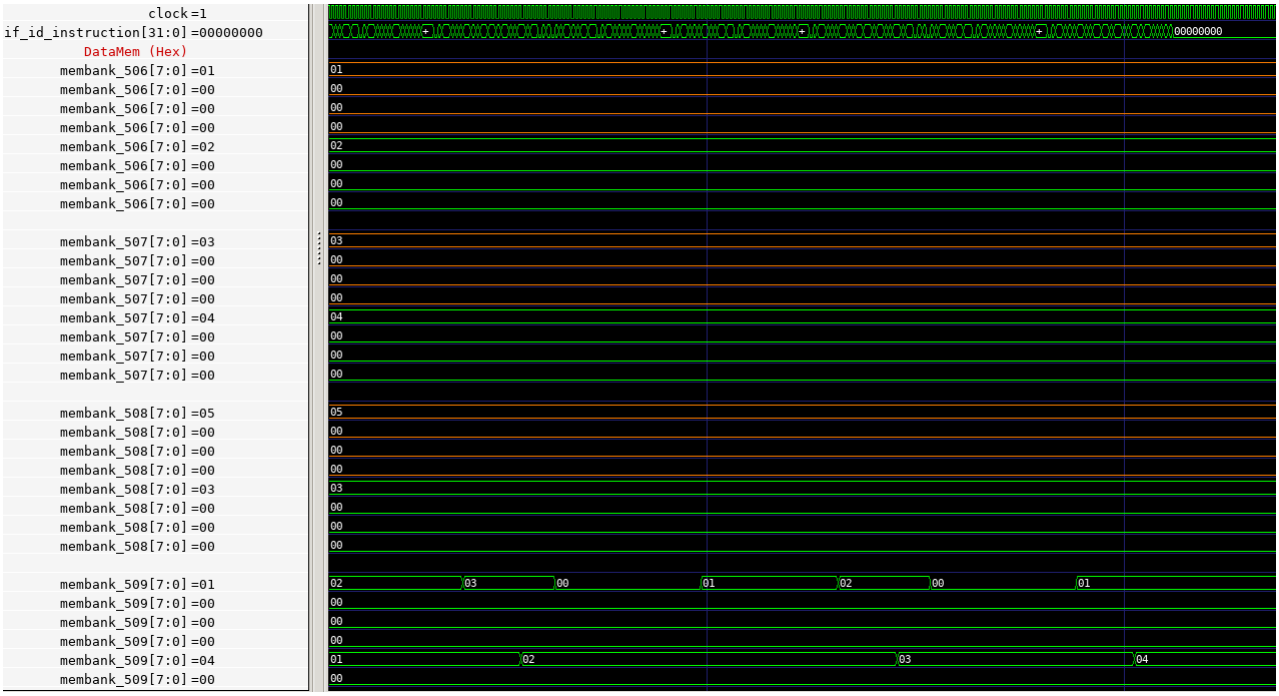

Figure 5: Occupied memory locations at the end of Bubble Sort execution. A Word(32-bit) occupies half of a membank and inside a Word the bits placed above in this picture are less significant than the ones below

derived from it in 3.3.
The instructions were obtained using the command line riscv64-unknown-elf-gcc -march=rv64im -c arithm.c, telling the compiler to use the M extension of the RV64I ISA (so the mul, div, rem, etc. instructions were used).
The first operation is a simple combination of addition/subtractions and multiplications/division/remainder operations to show the pipeline can handle the basics. The result is obtained correctly: see the program in the Appendix (operation **d** = ...), the data are int and Word instructions are used, so the result 0xFFFFFFF1 is 32-bit and is stored in a Word in the memory, see Fig.6, showing ALU_result (extended to 64 bits) and the membank (with the 32 bits stored).
The other operations test more delicate aspects of the micro-architecture.
Referring to the program in the Appendix there are:

e Division by zero. The result's every bit is set to 1: see Fig.7.

g1 There is no add unsigned operation in the RV64IM ISA so even though the C program specifies the unsigned declaration of f and g1, the compiler will use an ADDW instruction and the micro-architecture will interpret the numbers as signed so the sum will be 0: Fig.8.

g2 This time there's an overflow so the result in Fig.9 is effectively wrong (x800...0 is negative), the pipeline emits a warning at simulation time and the register for exception gets updated (a cycle later as mentioned before).

g3 This time it's an overflow in multiplication, since the result (g3 = 0x31FFFFFFFFFFFFF9C) needs more than 64-bits.
The MUL instruction returns the lower 64 bits of the 128 result register. The micro-architecture doesn't provide an overflow flag for multiplication, it's foreseen the use of a MULH instruction to obtain and check the high bits of the result register. A MULH instruction was added to the instruction list. Fig.10 shows the begin of the MUL operation and the end of the MUL, and it also shows the end of the MULH operation. Combining the two results, the expected result can be found. Note that the inputs of the ALU, the alu_operands in the figure, can change while the ALU is stalling for a mul/div but the meaningful ones have been initialized in the multiplier-divider at the beginning, as explained in the hardware description section, so what appears in the figures at the end of (or during) a mul operation is just garbage data.

g4 Next is an operation showing that an overflow can happen on addition/subtraction even with a Word operation. The ADD(I)W operation ignores the upper 32-bits and does a 32-bit addition, then sign extends the result. The 32-bit addition can overflow. See the program comments for details and Fig.11 showing the 'wrong' arithmetic result.

rm and r2m Similarly overflow can happen with a MULW instruction and it's the same situation as the previous MUL, just with 32-bits. This is rm. Using Double-word storage with the same data, as in r2m, makes the compiler use a 64-bit MUL and of course there are no issues. Again see program for numbers and comments and Fig.12 for the 'wrong' arithmetic result rm and the right result r2m.

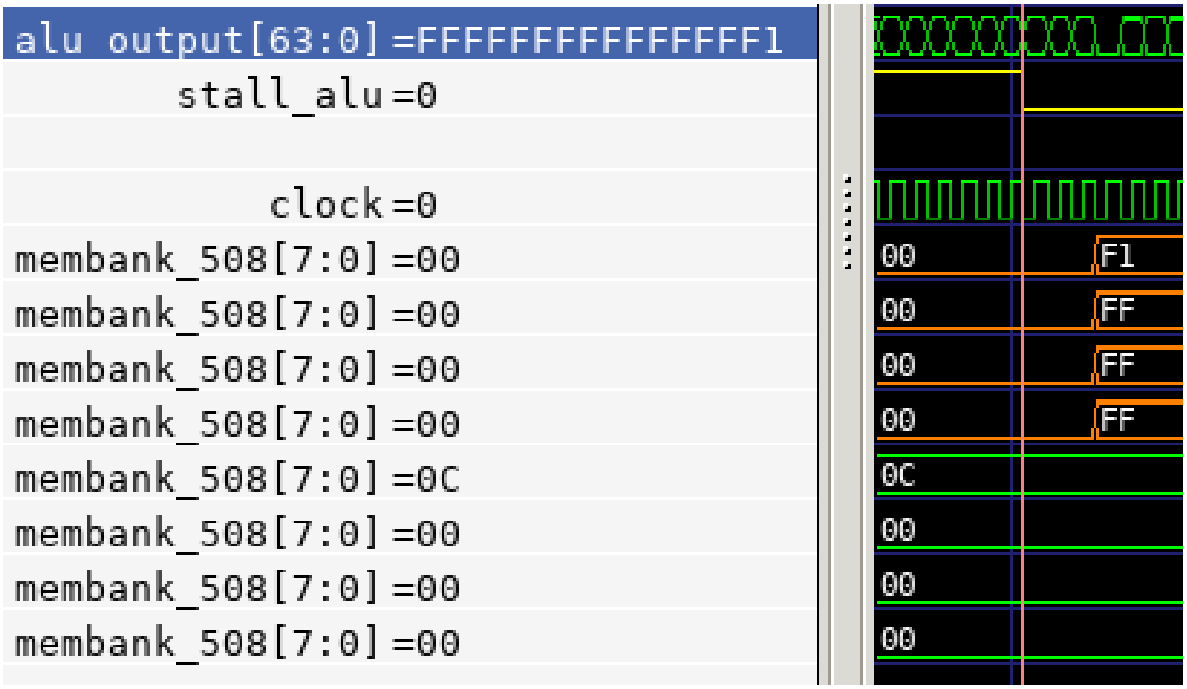

Figure 6: result d

Figure 7: result e

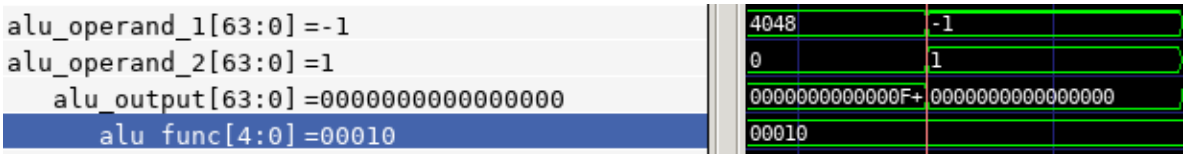

Figure 8: result g1

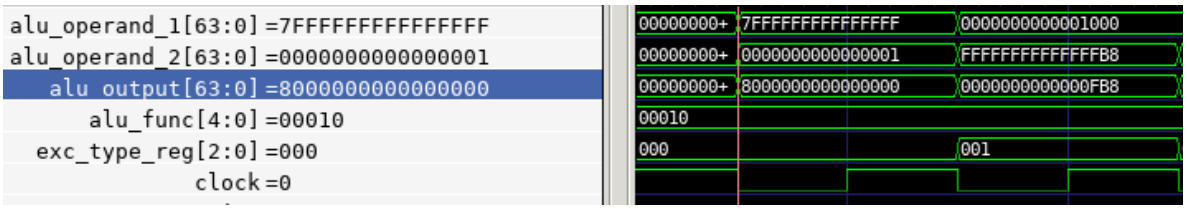

Figure 9: result g2

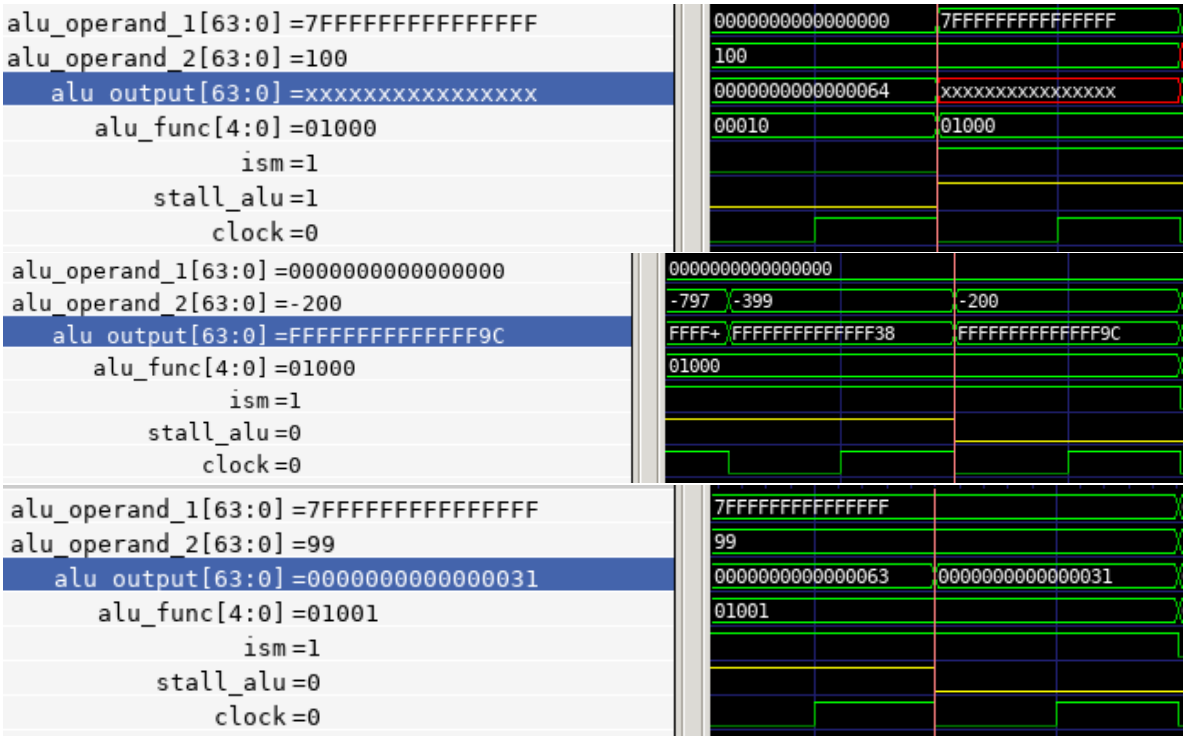

Figure 10: g3 operation begin and end of the MUL and end of the MULH

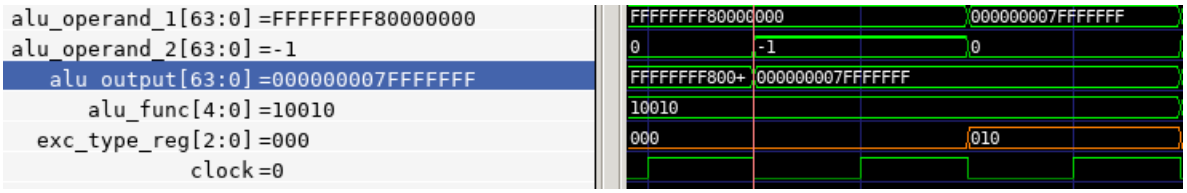

Figure 11: result g4

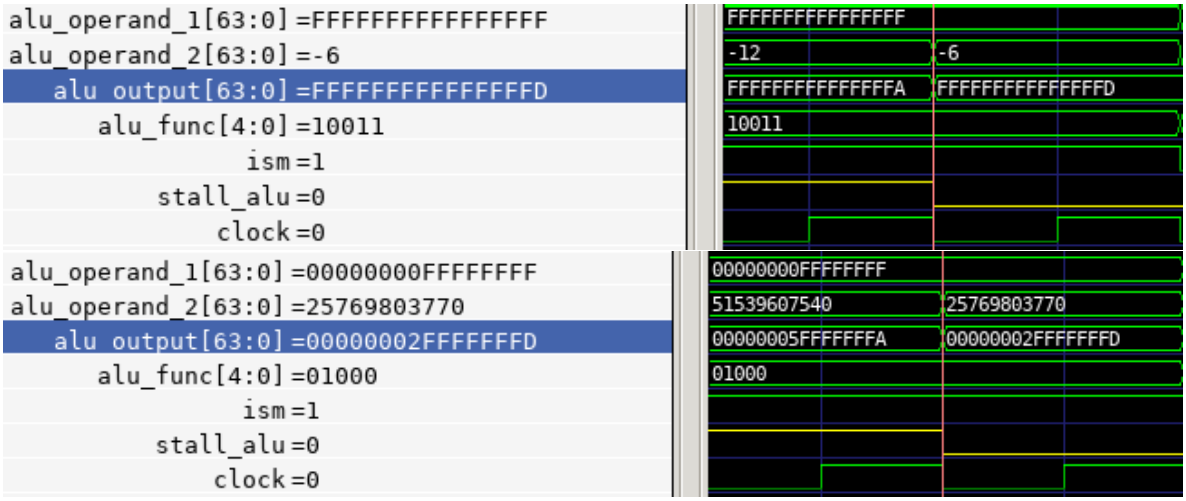

Figure 12: rm and r2m end result

3 Appendix

3.1 Instructions from bubble_sort.c program:

```
    X"fd010113",           --addi sp,sp,-48
    X"02813423",           --sd s0,40(sp)
    X"03010413",           --addi s0,sp,48
    X"00300793",           --li a5,3
    X"fcf42823",           --sw a5,-48(s0)
    X"00500793",           --li a5,5
    X"fcf42a23",           --sw a5,-44(s0)
    X"00100793",           --li a5,1
    X"fcf42c23",           --sw a5,-40(s0)
    X"00200793",           --li a5,2
    X"fcf42e23",           --sw a5,-36(s0)
    X"00400793",           --li a5,4
    X"fef42023",           --sw a5,-32(s0)
    X"fe042623",           --sw zero,-20(s0)
    X"06e0006f", --"00000110111000000000000001101111",  --  j .L2    --jal x0, 220  (immediate encode
    X"fe042423",           --sw zero,-24(s0)      --.L6
    X"0560006f", --"00000101011000000000000001101111" --  j .L3    --jal x0, 172
    X"fe842783",           --lw a5,-24(s0)        --.L5
    X"00279793",           --slli a5,a5,0x2
    X"ff078793",           --addi a5,a5,-16
    X"008787b3",           --add a5,a5,s0
    X"fe07a703",           --lw a4,-32(a5)
```

```
X"fe842783",           --lw a5,-24(s0)
X"0017879b",           --addiw a5,a5,1
X"0007879b",           --sext.w a5,a5
X"00279793",           --slli a5,a5,0x2
X"ff078793",           --addi a5,a5,-16
X"008787b3",           --add a5,a5,s0
X"fe07a783",           --lw a5,-32(a5)
X"02e7db63",  -- bge a5,a4,.L4
X"fe842783",           --lw a5,-24(s0)
X"00279793",           --slli a5,a5,0x2
X"ff078793",           --addi a5,a5,-16
X"008787b3",           --add a5,a5,s0
X"fe07a783",           --lw a5,-32(a5)
X"fef42223",           --sw a5,-28(s0)
X"fe842783",           --lw a5,-24(s0)
X"0017879b",           --addiw a5,a5,1
X"0007879b",           --sext.w a5,a5
X"00279793",           --slli a5,a5,0x2
X"ff078793",           --addi a5,a5,-16
X"008787b3",           --add a5,a5,s0
X"fe07a703",           --lw a4,-32(a5)
X"fe842783",           --lw a5,-24(s0)
X"00279793",           --slli a5,a5,0x2
X"ff078793",           --addi a5,a5,-16
X"008787b3",           --add a5,a5,s0
X"fee7a023",           --sw a4,-32(a5)
X"fe842783",           --lw a5,-24(s0)
X"0017879b",           --addiw a5,a5,1
X"0007879b",           --sext.w a5,a5
X"00279793",           --slli a5,a5,0x2
X"ff078793",           --addi a5,a5,-16
X"008787b3",           --add a5,a5,s0
X"fe442703",           --lw a4,-28(s0)
X"fee7a023",           --sw a4,-32(a5)
X"fe842783",           --lw a5,-24(s0)  --.L4
X"0017879b",           --addiw a5,a5,1
X"fef42423",           --sw a5,-24(s0)
X"00400793",           --li a5,4        --.L3
X"fec42703",           --lw a4,-20(s0)
X"40e787bb",           --subw a5,a5,a4
X"0007871b",           --sext.w a4,a5
X"fe842783",           --lw a5,-24(s0)
X"0007879b",           --sext.w a5,a5
X"fae7c0e3",  -- blt a5,a4,-96 <.L5>      -48*4=-192 --> -192/2=-96 bec then it gets shifted
X"fec42783",           --lw a5,-20(s0)
X"0017879b",           --addiw a5,a5,1
```

```
    X"fef42623",           --sw a5,-20(s0)
    X"fec42783",           --lw a5,-20(s0)  --.L2
    X"0007871b",           --sext.w a4,a5
    X"00300793",           --li a5,3
    X"f8e7d7e3", --bge a5,a4, -114 <.L6>       -57*4/2=-114
    X"00000793",           --li a5,0
    X"00078513",           --mv a0,a5
    X"02813403",           --ld s0,40(sp)
    X"03010113",           --addi sp,sp,48
    --X"00008067"          --ret --jalr x0, 0(x1)
    X"00000013"            --nop --addi x0, x0, 0
```

3.2 Code for arithm.c

```
#include <stdio.h>
int main(){

  int a = 3;
  int b = 4;
  int c;
  c = a*b; // c = 3*4 = 12
  b = a/c; // b = 3/12 = 0
  a = a%c; // a = 3%12 = 3

  int d = (a + (b+3)*c - (c+200)/a)/(3*b + c/5);
   // d = (3 + (0+3)*12 - (12+200)/3 ) / (3*0 + 12/5) =
   //   = (39 - 70) / 2 = -15 = 0xFFFFFFF1

  int e = -4/0;
  unsigned long int f = 0xFFFFFFFFFFFFFFFF;// -1 if signed, highest possible number
                                           // if unsigned
  unsigned long int g1 = f + 1; // there is not ADD Unsigned instruction in RV64I
                                // so f will be interepreted as -1 and g will just
                                // be 0
  signed long int h = 0x7FFFFFFFFFFFFFFF; // 64-bit highest possible 2's complement
                                          // signed number
  signed long int g2 = h + 1; // This time there should be an overflow

  signed long int g3 = h * 100; // This should overflow beyond 64 bits
                                // (g3 = 0x31FFFFFFFFFFFFF9C). The microarchitecture
                                // doesn't have an overflow signal for MUL.
                                // The MULH instruction should be used to check the
                                // bits beyond the 64th. I'll add a MULH to the
                                // instructions manually because the compiler doesn't
                                // produce it on it's own to check that the bits 0x31
```

```
                                // are there

  signed long int h2 = 0xFFFFFFFF80000000; // h2 contains the most negative 32-bit
                                           // number possible in its first half
                                           // (32-bits)...
  // ...for the following operation (g4) the compiler will use and ADDIW, that will
  // operate on these 32 bits, neglecting the others, but doing so there will
  // be an overflow in a Word operation:
  signed int g4 = h2 - 1; // it's int, not long int, so the compiler uses an ADDIW,
                          // not ADDI

  unsigned int om1 = 0xFFFFFFFF; // this is 32-bit lenght
  unsigned int om2 = 3;
  unsigned long int rm = om1*om2; // here I'm trying to declare rm long int (64-bit),
                                  // but the compiler still uses a MULW instruction,
                                  // so this operation will encounter the same overflow
                                  // situation as before (rm = 0x2FFFFFFFD > 32-bit).
                                  // A MUL instruction instead, would have produced an
                                  // output containing immediately the right result in
                                  // its least significant 64 bits.
  // This next attempt will use MUL and there won't be overflow problems,
  // but obviously it will consume more memory because I'm just using 64-bit operands
  // to host data that could fit in 32 bits:
  unsigned long long int o2m1 = 0xFFFFFFFF;
  unsigned long long int o2m2 = 3;
  unsigned long long int r2m = o2m1*o2m2;
  return 0;
}
```

3.3 instructions from arithm.c program

```
  X"f7010113",    --addi sp,sp,-144
  X"08813423",    --sd s0,136(sp)
  X"09010413",    --addi s0,sp,144
  X"00300793",    --li a5,3
  X"fef42623",    --sw a5,-20(s0)
  X"00400793",    --li a5,4
  X"fef42423",    --sw a5,-24(s0)
  X"fec42783",    --lw a5,-20(s0)
  X"00078713",    --mv a4,a5
  X"fe842783",    --lw a5,-24(s0)
  X"02f707bb",    --mulw a5,a4,a5
  X"fef42223",    --sw a5,-28(s0)
  X"fec42783",    --lw a5,-20(s0)
  X"00078713",    --mv a4,a5
  X"fe442783",    --lw a5,-28(s0)
```

```
X"02f747bb",    --divw a5,a4,a5
X"fef42423",    --sw a5,-24(s0)
X"fec42783",    --lw a5,-20(s0)
X"00078713",    --mv a4,a5
X"fe442783",    --lw a5,-28(s0)
X"02f767bb",    --remw a5,a4,a5
X"fef42623",    --sw a5,-20(s0)
X"fe842783",    --lw a5,-24(s0)
X"0037879b",    --addiw a5,a5,3
X"0007879b",    --sext.w a5,a5
X"fe442703",    --lw a4,-28(s0)
X"02f707bb",    --mulw a5,a4,a5
X"0007879b",    --sext.w a5,a5
X"fec42703",    --lw a4,-20(s0)
X"00f707bb",    --addw a5,a4,a5
X"0007871b",    --sext.w a4,a5
X"fe442783",    --lw a5,-28(s0)
X"0c87879b",    --addiw a5,a5,200
X"0007879b",    --sext.w a5,a5
X"fec42683",    --lw a3,-20(s0)
X"02d7c7bb",    --divw a5,a5,a3
X"0007879b",    --sext.w a5,a5
X"40f707bb",    --subw a5,a4,a5
X"0007871b",    --sext.w a4,a5
X"fe842783",    --lw a5,-24(s0)
X"00078693",    --mv a3,a5
X"00068793",    --mv a5,a3
X"0017979b",    --slliw a5,a5,0x1
X"00d787bb",    --addw a5,a5,a3
X"0007869b",    --sext.w a3,a5
X"fe442783",    --lw a5,-28(s0)
X"00078613",    --mv a2,a5
X"00500793",    --li a5,5
X"02f647bb",    --divw a5,a2,a5
X"0007879b",    --sext.w a5,a5
X"00f687bb",    --addw a5,a3,a5
X"0007879b",    --sext.w a5,a5
X"02f747bb",    --divw a5,a4,a5  -- last div in d = ...;
X"fef42023",    --sw a5,-32(s0)
X"ffc00793",    --li a5,-4
X"00000713",    --li a4,0
X"02e7c7bb",    --divw a5,a5,a4  -- e = -4/0;
X"fcf42e23",    --sw a5,-36(s0)
X"fff00793",    --li a5,-1
X"fcf43823",    --sd a5,-48(s0)
X"fd043783",    --ld a5,-48(s0)
```

```
X"00178793",    --addi a5,a5,1   -- g1 = f + 1;
X"fcf43423",    --sd a5,-56(s0)
X"fff00793",    --li a5,-1
X"0017d793",    --srli a5,a5,0x1
X"fcf43023",    --sd a5,-64(s0)
X"fc043783",    --ld a5,-64(s0)
X"00178793",    --addi a5,a5,1   -- g2 = h + 1;
X"faf43c23",    --sd a5,-72(s0)
X"fc043703",    --ld a4,-64(s0)
X"06400793",    --li a5,100
X"02f707b3",    --mul a5,a4,a5    -- g3 = h * 100;
X"06400813",    --li a6,100      --manually added
X"03071833",    --mulh a6,a4,a6   --manually added
X"faf43823",    --sd a5,-80(s0)
X"800007b7",    --lui a5,0x80000
X"faf43423",    --sd a5,-88(s0)
X"fa843783",    --ld a5,-88(s0)
X"0007879b",    --sext.w a5,a5
X"fff7879b",    --addiw a5,a5,-1 # 7fffffff <main+0x7fffffff> -- g4 = h2 - 1;
X"0007879b",    --sext.w a5,a5
X"faf42223",    --sw a5,-92(s0)
X"fff00793",    --li a5,-1
X"faf42023",    --sw a5,-96(s0)
X"00300793",    --li a5,3
X"f8f42e23",    --sw a5,-100(s0)
X"fa042783",    --lw a5,-96(s0)
X"00078713",    --mv a4,a5
X"f9c42783",    --lw a5,-100(s0)
X"02f707bb",    --mulw a5,a4,a5  -- rm = om1*om2;
X"0007879b",    --sext.w a5,a5
X"02079793",    --slli a5,a5,0x20
X"0207d793",    --srli a5,a5,0x20
X"f8f43823",    --sd a5,-112(s0)
X"fff00793",    --li a5,-1
X"0207d793",    --srli a5,a5,0x20
X"f8f43423",    --sd a5,-120(s0)
X"00300793",    --li a5,3
X"f8f43023",    --sd a5,-128(s0)
X"f8843703",    --ld a4,-120(s0)
X"f8043783",    --ld a5,-128(s0)
X"02f707b3",    --mul a5,a4,a5    -- r2m = o2m1*o2m2;
X"f6f43c23",    --sd a5,-136(s0)
X"00000793",    --li a5,0
X"00078513",    --mv a0,a5
X"08813403",    --ld s0,136(sp)
X"09010113",    --addi sp,sp,144
```

```
X"00000013"     -- nop --addi x0, x0, 0
--X"00008067"    --ret
```

References

[1] Michael Clark. *Risc-V Instructions table*. 2017. URL: https://github.com/michaeljclark/rv8/blob/master/doc/pdf/riscv-instructions.pdf.